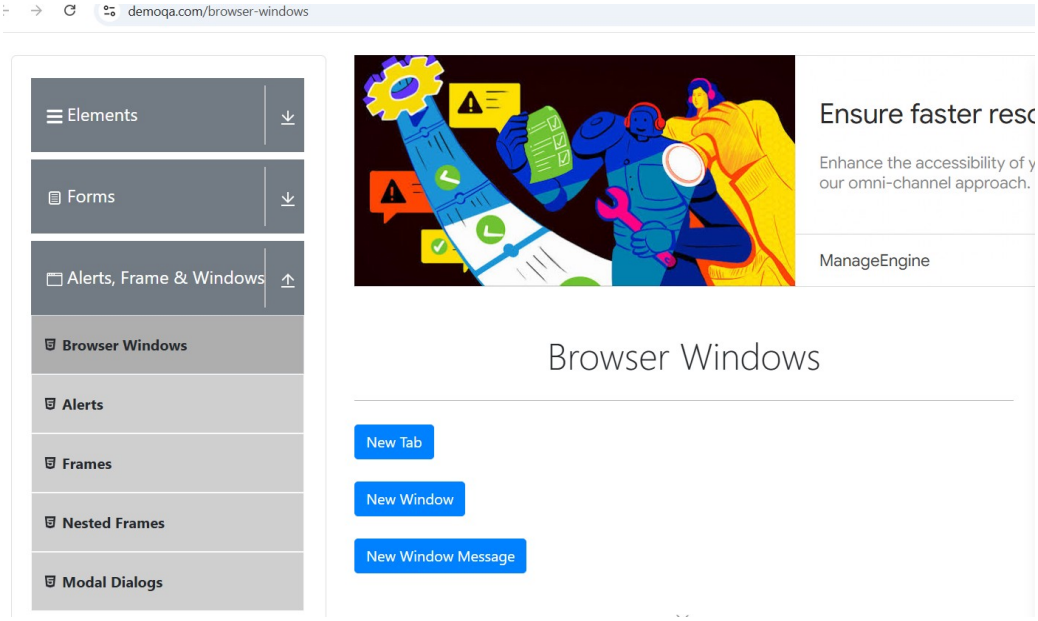
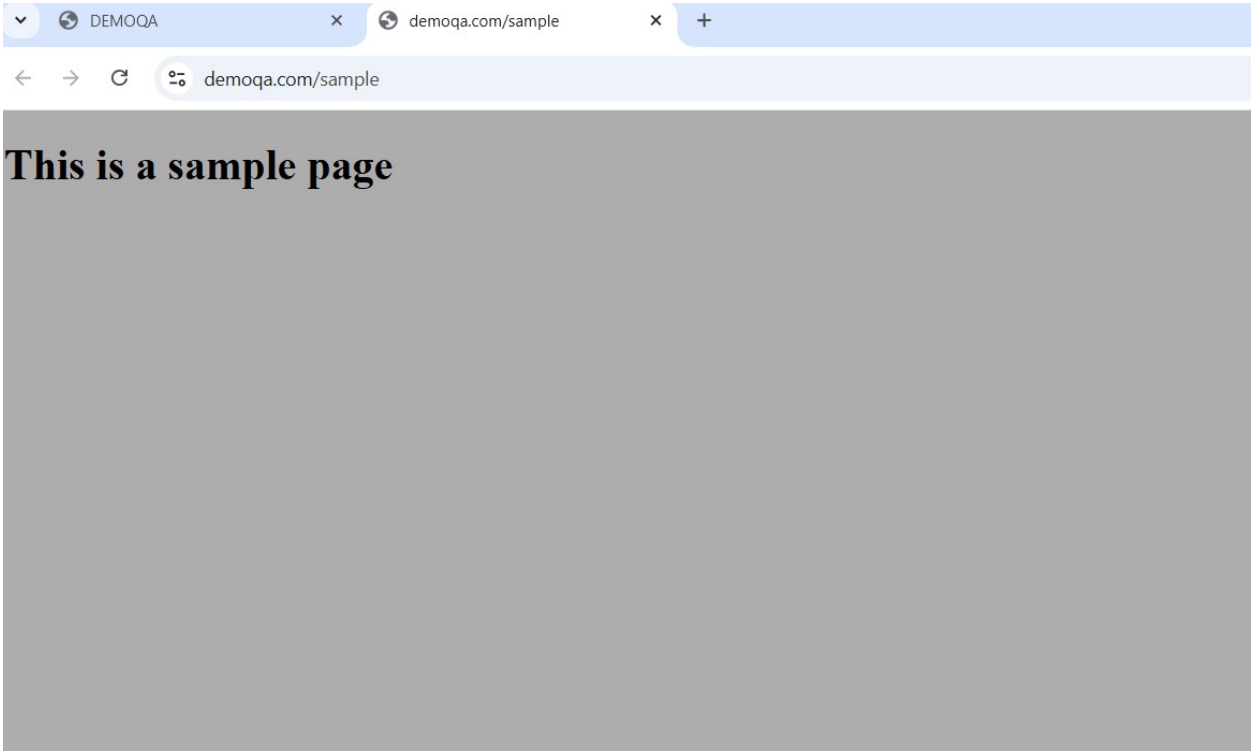
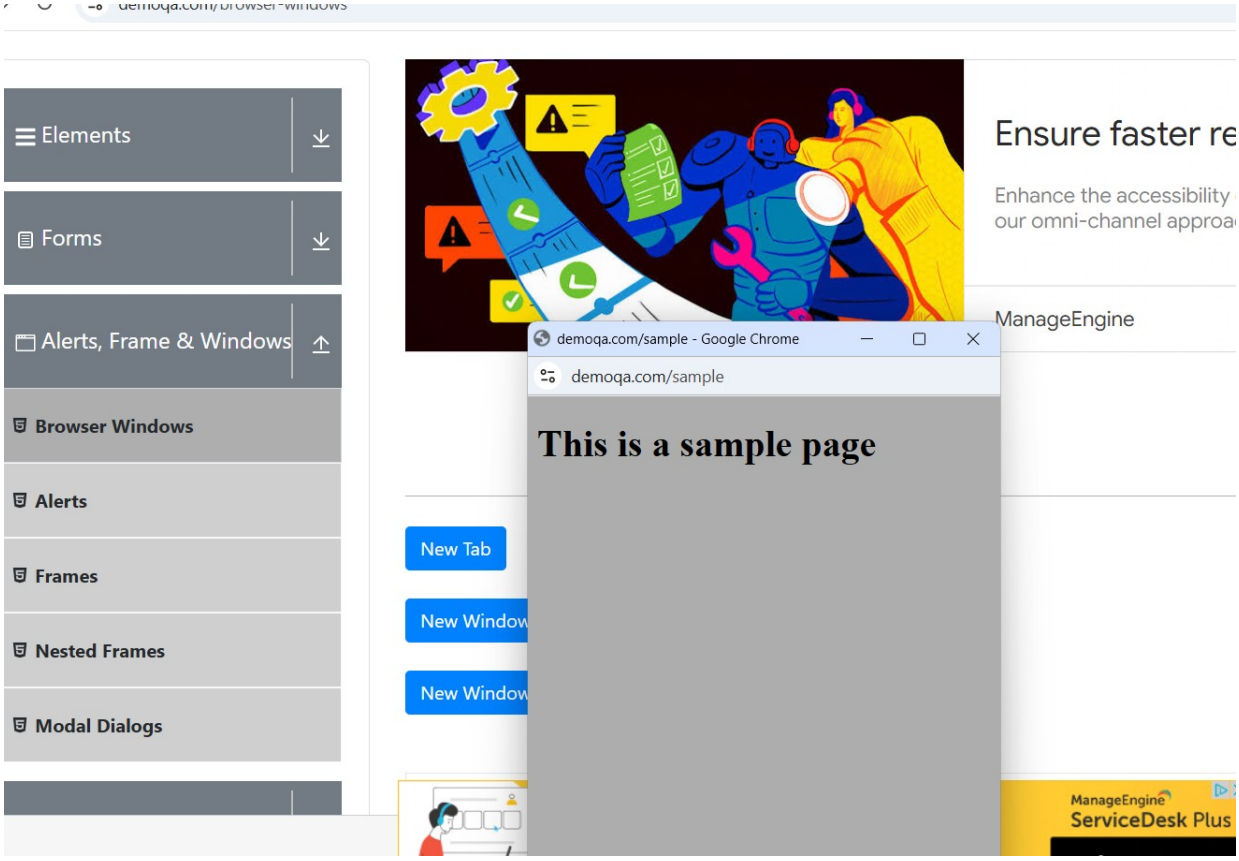


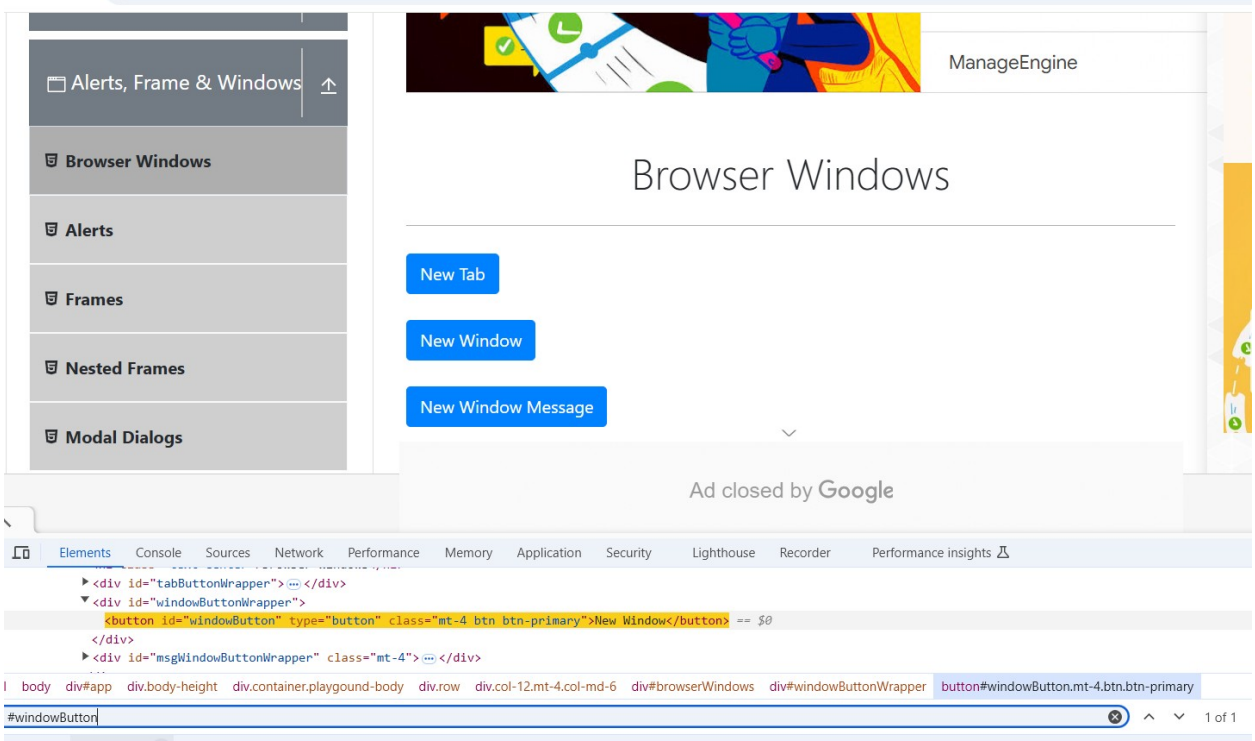
New Tab:



New Window:



Inspection od New Window Button by Id



The unique identifier (GUID)

In the context of web automation (e.g., Selenium), the unique identifier (GUID) for a browser window is referred to as the window handle.

What is a Window Handle?

- **A window handle is a unique string assigned to every browser window or tab in a web session.**
- **It acts as an identifier, allowing automation tools to interact with and switch between windows.**

```
The main window GUID is: DE256A8C6D2A83F8106D192A94A8A8C4
All Window GUIDS: [DE256A8C6D2A83F8106D192A94A8A8C4, 2C8935F34CEA99E7069F1F9DC598589A]
Switched to the window GUID: 2C8935F34CEA99E7069F1F9DC598589A
Current Page Title: DEMOQA

1 Scenarios (1 passed)
7 Steps (7 passed)
0m23.348s
```

This scenario automates the following workflow:

1. **Open a predefined browser window page.**
2. **Retrieve and store the unique identifier (GUID) of the main browser window.**
3. **Click a button to open a new browser window.**
4. **Switch context to the new window.**
5. **Validate the content of the new window.**

The automation validates that:

- **Windows can be properly identified and switched.**
- **Expected content is displayed on the new window.**

```
@windows
Feature: Browser Windows and Tabs Handling

  Scenario: Get the GUID of the main window
    Given I land on Demo QA Browser Window Page
    Then I can retrieve the GUID of the current window

    When I click on New Window button
    Then a new window is created and I switch to it
    And I can see the content on the new window as "This is a sample page"
```

```
package runner;

import io.cucumber.junit. Cucumber;
import io.cucumber.junit.CucumberOptions;
import org.junit.runner. RunWith;
@RunWith(Cucumber.class)
@CucumberOptions(
        features="src/test/resources/Features/loginPage.feature",
        glue = {"step_definitions"},
        dryRun= false ,
        tags= "@windows",
        plugin={"pretty", "html:target/cucumber-reports"},
        monochrome = false
)

public class TestRunner {
}
```

```
package pages;

import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import org.openqa.selenium.support.FindBy;
import org.openqa.selenium.support.PageFactory;

public class DemoQaBrowserWindowPage {
    private WebDriver driver;

    public DemoQaBrowserWindowPage(WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
```

```
    @FindBy(id = "windowButton")
    public WebElement newWindowButton;

    @FindBy(id = "sampleHeading")
    public WebElement contentElementNewWindow;

    public WebElement getNewWindowButton() {
        return newWindowButton;
    }

    public WebElement getContentElementNewWindow() {
        return contentElementNewWindow;
    }
}
```

This Java code defines a **Page Object Model (POM)** class for interacting with a specific page in a Selenium web automation framework. The class is named `DemoQaBrowserWindowPage` and encapsulates the web elements and their interactions on the **"Demo QA Browser Window" page**.

Here's a detailed explanation of the components:

---

1. Class Declaration

```
package pages;
```

- Declares the `pages` package, which organizes the code into a structured hierarchy. This is a common practice in larger projects to group related classes logically.

---

2. Instance Variable: `private WebDriver driver`

```
private WebDriver driver;
```

- This variable holds the reference to the **WebDriver** instance, which controls the browser.
- It is `private` to encapsulate the driver and ensure controlled access within this class.

---

3. Constructor

```
public DemoQaBrowserWindowPage(WebDriver driver) {
    this.driver = driver;
    PageFactory.initElements(driver, this);
}
```

- **Purpose:**
  Initializes the class with a WebDriver instance and sets up the web elements using the `PageFactory`.
- **Key Points:**
  1. `this.driver = driver;`
     Assigns the passed `WebDriver` instance to the local `driver` variable.
  2. `PageFactory.initElements(driver, this);`
     This initializes all the `@FindBy`-annotated web elements in the class.
     - `PageFactory` is part of Selenium's **Page Object Model (POM)** implementation.
     - It binds the web elements to their locators.

---

4. Web Elements

```
@FindBy(id = "windowButton")
public WebElement newWindowButton;

@FindBy(id = "sampleHeading")
public WebElement contentElementNewWindow;
```

- These are the **WebElements** representing specific elements on the "Demo QA Browser Window" page.
- **Annotations:**
  - `@FindBy`: Specifies how to locate the web elements.
  - `id = "windowButton"`: Locates the element with the `id` attribute `windowButton`.
  - `id = "sampleHeading"`: Locates the element with the `id` attribute `sampleHeading`.
- **Fields:**
  - `newWindowButton`: Represents the button that opens a new window.
  - `contentElementNewWindow`: Represents the content element (a heading or text) inside the new window.

---

5. Getter Methods

```
public WebElement getNewWindowButton() {
    return newWindowButton;
```

```
}

public WebElement getContentElementNewWindow() {
    return contentElementNewWindow;
}
```

- **Purpose:**
  These methods provide controlled access to the web elements.
- **Why Use Getter Methods?**
  - Encapsulation: Allows external classes to interact with the elements without directly exposing the fields.
  - Flexibility: If the underlying implementation changes (e.g., locators are updated), only the Page Object class needs modification, keeping the tests stable.

How This Class Is Used

This class is a **Page Object Model (POM)** representation of the "Demo QA Browser Window" page. It abstracts the interaction logic for this page, making test scripts cleaner and more maintainable.

For example:

1. **Initialization:**

   ```
   java
   Copy code
   WebDriver driver = new ChromeDriver();
   DemoQaBrowserWindowPage page = new DemoQaBrowserWindowPage(driver);
   ```

2. **Click the "New Window" Button:**

   ```
   java
   Copy code
   page.getNewWindowButton().click();
   ```

3. **Validate Content in the New Window:**

   ```
   java
   Copy code
   String content = page.getContentElementNewWindow().getText();
   Assert.assertEquals(content, "This is a sample page");
   ```

Why Use Page Object Model (POM)?

1. **Code Reusability:**
   Encapsulates element locators and actions, so they can be reused across multiple tests.
2. **Maintainability:**
   Changes in locators affect only the Page Object class, not all tests.
3. **Readability:**
   Simplifies test scripts by separating page interaction logic from test logic.

```
package step_definitions;

import io.cucumber.java.After;
import io.cucumber.java.Before;
import io.cucumber.java.Scenario;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import utilities.BrowserFactory;
import utilities.Driver;

import java.util.concurrent.TimeUnit;

public class Hooks {
    private WebDriver driver = null;

    @Before
    public void setUp() {
        // Corrected method call for BrowserFactory instance creation
        driver = BrowserFactory.createInstance();
        Driver.getInstance().setDriver(driver);
        driver = Driver.getInstance().getDriver();

        // Set up browser properties
        driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
        driver.manage().window().maximize();
    }

    @After
    public void tearDown(Scenario scenario) {
        // Capture screenshot if scenario fails
        if (scenario.isFailed()) {
            final byte[] screenshot = ((TakesScreenshot)
Driver.getInstance().getDriver()).getScreenshotAs(OutputType.BYTES);
            scenario.attach(screenshot, "image/png", "Screenshot on
failure");
```

```
        }

        // Remove the driver instance
        Driver.getInstance().removeDriver();
    }
}
```

This code defines **hooks** in a Cucumber-based Selenium test framework. Hooks are special methods in Cucumber annotated with `@Before` and `@After` that execute before and after each test scenario. They are commonly used for test setup and teardown activities, ensuring consistent test environments.

Here's a detailed explanation:

---

1. Package Declaration

```
package step_definitions;
```

- Organizes this class in the `step_definitions` package alongside other step definition or supporting classes.

---

2. Imports

```
import io.cucumber.java.After;
import io.cucumber.java.Before;
import io.cucumber.java.Scenario;
import org.openqa.selenium.OutputType;
import org.openqa.selenium.TakesScreenshot;
import org.openqa.selenium.WebDriver;
import utilities.BrowserFactory;
import utilities.Driver;
```

- **Key Libraries:**
  - `@Before` and `@After`: Cucumber annotations for hooks.
  - `Scenario`: Represents the current Cucumber scenario; provides details like status (pass/fail).
  - `TakesScreenshot`: Selenium interface for capturing screenshots.
  - `Driver`: A utility class to manage a singleton WebDriver instance.
  - `BrowserFactory`: A utility class for creating a specific browser instance.

---

3. Class Fields

```
private WebDriver driver = null;
```

- A private variable to hold the WebDriver instance. This is used to manage the browser during the test lifecycle.

---

4. The `@Before` Hook

```
@Before
public void setUp() {
    driver = BrowserFactory.createInstance(); // Creates a new browser instance
    Driver.getInstance().setDriver(driver);   // Stores the instance in a thread-safe
Driver manager
    driver = Driver.getInstance().getDriver(); // Retrieves the WebDriver instance

    // Set browser properties
    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS); // Implicit wait
    driver.manage().window().maximize();                             // Maximizes the
browser window
}
```

- **Purpose:** Sets up the WebDriver and browser environment before executing each scenario.
- **Detailed Steps:**
  1. **Create a Browser Instance:**
     `BrowserFactory.createInstance()` initializes a WebDriver for a specific browser (e.g., Chrome or Firefox).
  2. **Manage Driver Singleton:**
     - `Driver.getInstance()` ensures there's a single thread-safe WebDriver instance.
     - This is useful when running parallel tests to avoid conflicts between threads.
  3. **Configure Browser Settings:**
     - **Implicit Wait:** Waits up to 10 seconds for elements to appear on the DOM before throwing exceptions.
     - **Maximize Window:** Maximizes the browser to ensure all elements are visible.

---

5. The `@After` Hook

```
@After
public void tearDown(Scenario scenario) {
    if (scenario.isFailed()) {
```

```
        // Capture screenshot if the scenario fails
        final byte[] screenshot = ((TakesScreenshot) Driver.getInstance().getDriver())
            .getScreenshotAs(OutputType.BYTES);
        scenario.attach(screenshot, "image/png", "Screenshot on failure");
    }

    // Clean up the WebDriver instance
    Driver.getInstance().removeDriver();
}
```

- **Purpose:** Cleans up resources after executing a scenario, and captures screenshots for failed scenarios.
- **Detailed Steps:**
  1. **Check Scenario Status:**
     `scenario.isFailed()` returns `true` if the scenario failed. This ensures screenshots are only taken for failed tests.
  2. **Capture Screenshot:**
     - `(TakesScreenshot) driver`: Casts the WebDriver instance to `TakesScreenshot`.
     - `getScreenshotAs(OutputType.BYTES)`: Captures the screenshot as a byte array.
     - `scenario.attach(...)`: Attaches the screenshot to the scenario report in a format supported by Cucumber (e.g., `image/png`).
  3. **Remove Driver Instance:**
     - `Driver.getInstance().removeDriver()` ensures the WebDriver instance is properly closed and removed from memory after the test.

---

6. Supporting Utilities

`Driver` Class (Singleton)

- Manages a thread-safe WebDriver instance.
- Key methods:
  - `setDriver(driver)`: Stores the driver instance.
  - `getDriver()`: Retrieves the stored driver instance.
  - `removeDriver()`: Quits the browser and removes the instance.

`BrowserFactory` Class

- Creates a WebDriver instance for a specified browser (e.g., Chrome or Firefox). Example:

```
public class BrowserFactory {
    public static WebDriver createInstance() {
        // Logic to initialize a specific browser instance
        return new ChromeDriver(); // Default: Chrome
```

```
    }
}
```

How It Fits in the Test Flow

1. **Before Each Scenario:**
   - A WebDriver instance is created, configured, and made ready for test execution.
2. **Test Execution:**
   - The WebDriver interacts with the browser based on step definitions.
3. **After Each Scenario:**
   - If the scenario fails, a screenshot is captured for debugging.
   - The WebDriver is closed and resources are cleaned up to prevent memory leaks.

Key Benefits

1. **Reusable Setup and Teardown:**
   Centralizes the browser setup and cleanup process for all scenarios, avoiding redundancy.
2. **Improved Debugging with Screenshots:**
   Screenshots for failed scenarios help identify issues quickly.
3. **Scalable for Parallel Execution:**
   The `Driver` utility ensures thread-safe WebDriver management, critical for parallel testing.

Possible Enhancements

1. **Parameterize Browser Type:**
   Allow dynamic selection of the browser (e.g., via a configuration file or environment variable):

```
public static WebDriver createInstance(String browser) {
    switch (browser.toLowerCase()) {
        case "firefox":
            return new FirefoxDriver();
        case "chrome":
        default:
            return new ChromeDriver();
    }
}
```

2. **Replace Implicit Wait with Explicit Waits:**
   Use **explicit waits** for more robust handling of dynamic content:

```
new WebDriverWait(driver, Duration.ofSeconds(10))
    .until(ExpectedConditions.visibilityOfElementLocated(By.id("example")));
```

```
package utilities;

import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxOptions;

import java.util.HashMap;

public class BrowserFactory {

    public static WebDriver createInstance() {
        WebDriver driver = null;
        try {
            if (driver == null) {
                if (System.getProperty("browser") == null) {
                    WebDriverManager.chromedriver().setup();
                    ChromeOptions options = new ChromeOptions();
                    HashMap<String, Object> chromePrefs = new
HashMap<>();
                    chromePrefs.put("download.default_directory",
System.getProperty("user.dir") + "\\src\\test\\resources\\testData\\
Downloads");
                    options.addArguments("--ignore-ssl-errors=yes");
                    options.addArguments("--ignore-certificate-errors");
                    options.setExperimentalOption("prefs", chromePrefs);
                    driver = new ChromeDriver(options);
                } else {
                    switch (System.getProperty("browser")) {
                        case "chrome-headless":
                            WebDriverManager.chromedriver().setup();
                            ChromeOptions headlessOptions = new
ChromeOptions();
                            headlessOptions.addArguments("--headless");
                            headlessOptions.addArguments("--disable-
gpu");
                            driver = new ChromeDriver(headlessOptions);
```

```
                            break;

                        case "firefox-headless":
                            WebDriverManager.firefoxdriver().setup();
                            FirefoxOptions firefoxHeadlessOptions = new
FirefoxOptions();
                            firefoxHeadlessOptions.addArguments("--
headless");
                            driver = new
FirefoxDriver(firefoxHeadlessOptions);
                            break;

                        // Add cases for other browsers as required
                        default:
                            WebDriverManager.chromedriver().setup();
                            driver = new ChromeDriver();
                            break;
                    }
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
        return driver;
    }
}
```

This class, `BrowserFactory`, provides a utility to create and configure WebDriver instances for browser automation in Selenium. It uses the **Factory Design Pattern** to centralize browser setup logic, making it reusable and flexible. It also incorporates **dynamic browser selection** using system properties, with a default fallback to Chrome.

Here's a detailed explanation:

---

1. Package Declaration

```
package utilities;
```

- Groups this class under the `utilities` package, which typically contains helper or utility classes used across the test framework.

---

2. Imports

```
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import org.openqa.selenium.chrome.ChromeOptions;
import org.openqa.selenium.firefox.FirefoxDriver;
import org.openqa.selenium.firefox.FirefoxOptions;

import java.util.HashMap;
```

- **Key Libraries:**
  - **`WebDriverManager`:** Simplifies browser driver setup (e.g., `chromedriver`, `geckodriver`) by automatically managing driver binaries.
  - **`WebDriver`, `ChromeDriver`, `FirefoxDriver`:** Selenium WebDriver classes for browser automation.
  - **`ChromeOptions`, `FirefoxOptions`:** Used to configure browser-specific settings and capabilities.
  - **`HashMap`:** A collection to manage preferences for Chrome (e.g., download directory).

---

3. Method Overview:

```
public static WebDriver createInstance()
```

- A **static method** that initializes and returns a WebDriver instance based on:
  1. **System Property `browser`:** Allows users to specify the desired browser via runtime configuration.
  2. **Default Fallback:** If no browser is specified, Chrome is initialized by default.

---

4. Logic Breakdown

a. WebDriver Initialization

```
if (driver == null)
```

- Ensures that no new instance is created if one already exists, helping to avoid resource duplication.

---

b. Default Browser Setup (Chrome)

```
if (System.getProperty("browser") == null) {
    WebDriverManager.chromedriver().setup();
    ChromeOptions options = new ChromeOptions();
    HashMap<String, Object> chromePrefs = new HashMap<>();
    chromePrefs.put("download.default_directory", System.getProperty("user.dir") + "\\
src\\test\\resources\\testData\\Downloads");
    options.addArguments("--ignore-ssl-errors=yes");
    options.addArguments("--ignore-certificate-errors");
    options.setExperimentalOption("prefs", chromePrefs);
    driver = new ChromeDriver(options);
}
```

- **When no browser is specified** (`System.getProperty("browser") == null`), Chrome is initialized with:
  - **WebDriverManager Setup:**
    Automatically downloads the correct `chromedriver` version based on the local Chrome installation.
  - **Chrome Preferences:**
    Configures:
    - **Default Download Directory:** Ensures all downloads go to a specific folder (`testData/Downloads`).
    - **SSL and Certificate Handling:** Suppresses warnings/errors for self-signed certificates or insecure connections.
  - **`ChromeDriver` Instance:** A new WebDriver instance is created with the configured options.

---

c. Dynamic Browser Selection

```
switch (System.getProperty("browser"))
```

- Based on the value of the `browser` system property, different WebDriver instances are created.
  For example:
  - **`chrome-headless`:**
    - Chrome runs in headless mode (no GUI).
    - Useful for CI/CD pipelines and running tests on headless servers.
    - Options:
      - `--headless`: Enables headless mode.
      - `--disable-gpu`: Disables GPU usage to avoid rendering issues.
  - **`firefox-headless`:**
    - Similar to Chrome but uses Firefox in headless mode.
- **Default Case:**
  - If the specified browser is unrecognized, Chrome is used as a fallback.

---

d. Exception Handling

```
catch (Exception e) {
    e.printStackTrace();
}
```

- Captures any runtime exceptions (e.g., driver setup issues) and prints the stack trace to help with debugging.

---

5. How It Works in a Test Framework

1. **Browser Selection via System Property:**
    - Users specify the browser at runtime using a JVM argument:

      ```
      bash
      Copy code
      mvn test -Dbrowser=chrome-headless
      ```

    - If no browser is specified, Chrome with default settings is used.
2. **WebDriver Creation:**
    - A configured WebDriver instance is created and returned to the calling test class.
3. **Thread-Safe Usage:**
    - This method can be combined with a `Driver` singleton class to ensure thread-safe WebDriver management.

---

6. Example Usage

Here's how this `BrowserFactory` might be used in the framework:

Test Initialization

```
public class Driver {
    private static ThreadLocal<WebDriver> driver = new ThreadLocal<>();

    public static WebDriver getDriver() {
        if (driver.get() == null) {
            driver.set(BrowserFactory.createInstance());
        }
        return driver.get();
    }

    public static void removeDriver() {
        if (driver.get() != null) {
            driver.get().quit();
```

```
            driver.remove();
        }
    }
}
```

In a Hook Class (`@Before`)

```
@Before
public void setUp() {
    WebDriver driver = Driver.getDriver(); // Initialize driver via BrowserFactory
    driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
    driver.manage().window().maximize();
}
```

---

7. Advantages

1. **Flexibility:**
   - Supports multiple browsers and configurations.
   - Can easily add new browser setups (e.g., Edge, Safari).
2. **Centralized Management:**
   - Simplifies browser setup and configuration across the framework.
3. **Dynamic Selection:**
   - Allows testers to switch browsers at runtime without modifying the code.
4. **Headless Mode:**
   - Enables running tests in environments without a GUI, like CI/CD pipelines.

---

8. Potential Improvements

1. **Enhanced Configuration Management:**
   - Use a configuration file (e.g., `config.properties`) or environment variables to manage browser settings instead of hardcoding.
2. **Add Support for More Browsers:**
   - Include drivers for Edge, Safari, or other popular browsers.
3. **Replace `catch` Block with Logging:**
   - Use a logging framework like Log4j or SLF4J to record errors.
4. **Improve Download Directory Management:**
   - Make the download directory path configurable instead of hardcoding it.

```
package utilities;

import org.openqa.selenium.WebDriver;
```

```
public class Driver {

    private Driver(){}

    private static Driver driverInstance=new Driver();

    public static Driver getInstance(){
        return driverInstance;
    }

    ThreadLocal<WebDriver> threadLocalDriver=new
ThreadLocal<WebDriver>();

    public WebDriver getDriver(){
        WebDriver driver = threadLocalDriver.get();
        return driver;
    }

    public  void setDriver(WebDriver driverParameter){
        // set the parameter driver into threadLocal
        threadLocalDriver.set(driverParameter);
    }

    public void removeDriver(){
        WebDriver driver = threadLocalDriver.get();
        driver.quit();
        threadLocalDriver.remove();
    }
}
```

This `Driver` class is a **singleton utility** for managing WebDriver instances in a thread-safe manner. It ensures each thread in a parallel test execution gets its own WebDriver instance, avoiding conflicts between test cases running concurrently.

Here's a detailed breakdown:

---

1. Package Declaration

```
package utilities;
```

- Groups this class in the `utilities` package, which typically contains reusable helper classes.

---

2. Singleton Design Pattern

Private Constructor

```
private Driver(){}
```

- Prevents instantiation of the `Driver` class from outside, ensuring only one instance exists.

Static Instance

```
private static Driver driverInstance = new Driver();
```

- A single instance of the class is created and stored statically for global access.

Global Access

```
public static Driver getInstance() {
    return driverInstance;
}
```

- Provides controlled access to the singleton instance of `Driver`.

---

3. Thread-Local WebDriver Management

ThreadLocal Variable

```
ThreadLocal<WebDriver> threadLocalDriver = new ThreadLocal<>();
```

- Ensures each thread running a test gets its own WebDriver instance:
  - `ThreadLocal` maintains a separate copy of the WebDriver for each thread.
  - Prevents test cases from interfering with each other in parallel execution environments.

---

4. WebDriver Operations

Getter: Retrieving the Driver

```
public WebDriver getDriver() {
    WebDriver driver = threadLocalDriver.get();
    return driver;
```

```
}
```

- Retrieves the WebDriver instance specific to the current thread.

Setter: Setting the Driver

```
public void setDriver(WebDriver driverParameter) {
    threadLocalDriver.set(driverParameter);
}
```

- Assigns a WebDriver instance to the current thread by setting it in `ThreadLocal`.

Removing the Driver

```
public void removeDriver() {
    WebDriver driver = threadLocalDriver.get(); // Retrieve the current thread's
WebDriver
    driver.quit();                               // Close the browser
    threadLocalDriver.remove();                 // Remove the instance from ThreadLocal
}
```

- Safely removes the WebDriver instance associated with the current thread:
    1. **`quit`:** Closes the browser window and ends the WebDriver session.
    2. **`remove`:** Clears the `ThreadLocal` variable to free memory.

---

5. Why Use This Design?

1. **Thread-Safe Parallel Execution:**
    - In parallel testing, multiple threads execute concurrently, and each thread needs its own WebDriver instance.
    - `ThreadLocal` ensures that each thread uses its dedicated WebDriver instance without interference.
2. **Singleton with Thread Safety:**
    - While `Driver` is a singleton, it uses `ThreadLocal` to handle multiple WebDriver instances safely.
3. **Global Accessibility:**
    - The singleton design allows any class in the framework to access the `Driver` instance via `Driver.getInstance()`.

---

6. Usage in Framework

Setting the Driver

Called when initializing the WebDriver, usually in a `@Before` hook:

```
Driver.getInstance().setDriver(BrowserFactory.createInstance());
```

- The browser is initialized by `BrowserFactory`, and the instance is assigned to the current thread using `setDriver`.

Retrieving the Driver

Used in test steps or utilities to interact with the browser:

```
WebDriver driver = Driver.getInstance().getDriver();
driver.get("https://example.com");
```

Removing the Driver

Called in a `@After` hook to clean up after a test:

```
Driver.getInstance().removeDriver();
```

---

7. Example Workflow

1. **Test Initialization:**
   - A new WebDriver instance is created by `BrowserFactory` and assigned to the thread using `Driver.setDriver`.
2. **Test Execution:**
   - Each test retrieves its thread-local WebDriver instance via `Driver.getDriver()`.
3. **Test Cleanup:**
   - The WebDriver session is closed and removed using `Driver.removeDriver`.

---

8. Key Advantages

1. **Thread Safety:**
   - Isolates WebDriver instances for parallel tests, avoiding session overlap or race conditions.
2. **Memory Efficiency:**
   - Frees up memory by removing WebDriver instances from `ThreadLocal` after test execution.

3. **Scalability:**
    - Allows seamless integration with parallel test execution frameworks like TestNG or JUnit.
4. **Global Accessibility:**
    - Any test class or utility can access the WebDriver instance via `Driver.getInstance()`.

---

9. Potential Enhancements

1. **Exception Handling in `removeDriver`:**
    - Add error handling to manage cases where `quit()` might fail:

      ```
      java
      Copy code
      try {
          driver.quit();
      } catch (Exception e) {
          e.printStackTrace();
      } finally {
          threadLocalDriver.remove();
      }
      ```

2. **Configuration for Multiple Browsers:**
    - Integrate `BrowserFactory` to dynamically set the WebDriver based on runtime browser selection.
3. **Logging:**
    - Use a logging framework (e.g., Log4j) to log driver actions and cleanup steps.

---

10. Conclusion

This `Driver` class is a critical part of the test framework, enabling robust and thread-safe WebDriver management for parallel test execution. Its combination of the **Singleton** and **ThreadLocal** design patterns ensures scalability, maintainability, and reliability in browser automation testing.

```
package utilities;
import io.github.bonigarcia.wdm.WebDriverManager;
import org.openqa.selenium. WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
public class DriverFactory {
```

```
    private static WebDriver driver;
    public static WebDriver getDriver () {
        if (driver == null) {
            WebDriverManager.chromedriver().setup();
            driver = new ChromeDriver();
        }
        return driver;
    }
        public static void closeDriver () {
            if (driver != null) {
                driver.close();
                driver = null;
            }
        }
    }
```

The `DriverFactory` class is a **utility class** designed to manage the lifecycle of a single `WebDriver` instance in a Selenium test framework. It follows a **lazy initialization** approach to ensure the `WebDriver` instance is created only when it's needed. Here's a detailed explanation of the code:

---

1. Package Declaration

```
package utilities;
```

- Groups this class under the `utilities` package, typically containing helper classes.

---

2. Class-Level Variables

```
private static WebDriver driver;
```

- **Static Variable:**
  - Stores the single `WebDriver` instance.
  - Being `static`, it ensures that the same instance is accessible throughout the application.
- **Purpose:** Prevent multiple instances of `WebDriver`, which could lead to resource conflicts.

---

3. Singleton-Like `getDriver` Method

```
public static WebDriver getDriver() {
    if (driver == null) {
        WebDriverManager.chromedriver().setup();
        driver = new ChromeDriver();
    }
    return driver;
}
```

Explanation:

1. **Lazy Initialization:**
    - The `WebDriver` instance is created only when `getDriver` is called for the first time (`driver == null`).
    - Ensures efficient use of resources, avoiding premature browser initialization.
2. **WebDriverManager Integration:**
    - `WebDriverManager.chromedriver().setup();`
        - Automatically downloads and sets up the ChromeDriver binary compatible with the Chrome browser version on the machine.
3. **Driver Instantiation:**
    - A new instance of `ChromeDriver` is created if `driver` is `null`.
4. **Reusability:**
    - Once initialized, the same `WebDriver` instance is returned on subsequent calls to `getDriver`.

---

4. Driver Cleanup with `closeDriver` Method

```
public static void closeDriver() {
    if (driver != null) {
        driver.close();
        driver = null;
    }
}
```

Explanation:

1. **Driver Closure:**
    - `driver.close();`
        - Closes the current browser window.
        - Does **not** terminate the WebDriver session if multiple windows are open.
2. **Reset `driver`:**

    - `driver = null;`
        - Sets the `driver` variable back to `null`, ensuring the next call to `getDriver` creates a new instance.
3. **Safety Check:**
    - Prevents `NullPointerException` by checking if `driver != null` before attempting to close the driver.

---

5. Key Characteristics

Singleton-Like Behavior:

- Ensures only one `WebDriver` instance exists throughout the test session.

Resource Management:

- Avoids excessive resource usage by initializing the driver only when required (`lazy initialization`) and cleaning it up when not needed.

Encapsulation:

- Hides the complexity of `WebDriver` initialization and cleanup, making the class easy to use for test developers.

---

6. Usage in Test Framework

1. **Driver Initialization:**
    - Called in the `@Before` method of test hooks or test setup:

      ```
      WebDriver driver = DriverFactory.getDriver();
      driver.get("https://example.com");
      ```

2. **Driver Cleanup:**
    - Called in the `@After` method of test hooks or test teardown:

      ```
      DriverFactory.closeDriver();
      ```

---

7. Limitations

1. **Thread Safety:**
    - The `DriverFactory` class is **not thread-safe**, meaning parallel tests can cause conflicts as all threads would share the same `driver` instance.

    - To address this, use `ThreadLocal<WebDriver>` (like in the `Driver` class discussed earlier).
2. **Browser Flexibility:**
    - This implementation is limited to ChromeDriver. Adding support for other browsers requires modifications.

---

8. Enhancements

Thread-Safe Singleton:

- Use `ThreadLocal<WebDriver>` for managing WebDriver instances in parallel tests:

```
private static ThreadLocal<WebDriver> threadLocalDriver = new ThreadLocal<>();

public static WebDriver getDriver() {
    if (threadLocalDriver.get() == null) {
        WebDriverManager.chromedriver().setup();
        threadLocalDriver.set(new ChromeDriver());
    }
    return threadLocalDriver.get();
}

public static void closeDriver() {
    if (threadLocalDriver.get() != null) {
        threadLocalDriver.get().close();
        threadLocalDriver.remove();
    }
}
```

Dynamic Browser Support:

- Add support for multiple browsers (e.g., Firefox, Edge) based on system properties:

```
public static WebDriver getDriver() {
    if (driver == null) {
        String browser = System.getProperty("browser", "chrome");
        switch (browser.toLowerCase()) {
            case "firefox":
                WebDriverManager.firefoxdriver().setup();
                driver = new FirefoxDriver();
                break;
            case "edge":
                WebDriverManager.edgedriver().setup();
                driver = new EdgeDriver();
                break;
            default:
                WebDriverManager.chromedriver().setup();
                driver = new ChromeDriver();
```

```
            }
        }
        return driver;
    }
```

---

9. Summary

The `DriverFactory` class simplifies WebDriver management by implementing a singleton pattern to reuse a single WebDriver instance. While it's ideal for simple, sequential tests, it lacks thread safety for parallel execution. By introducing thread-local variables and multi-browser support, it can be adapted for advanced testing needs.

qa_configs.properties FILE

browserType=CHROME
username=yoll-student
password=Bootcamp5#

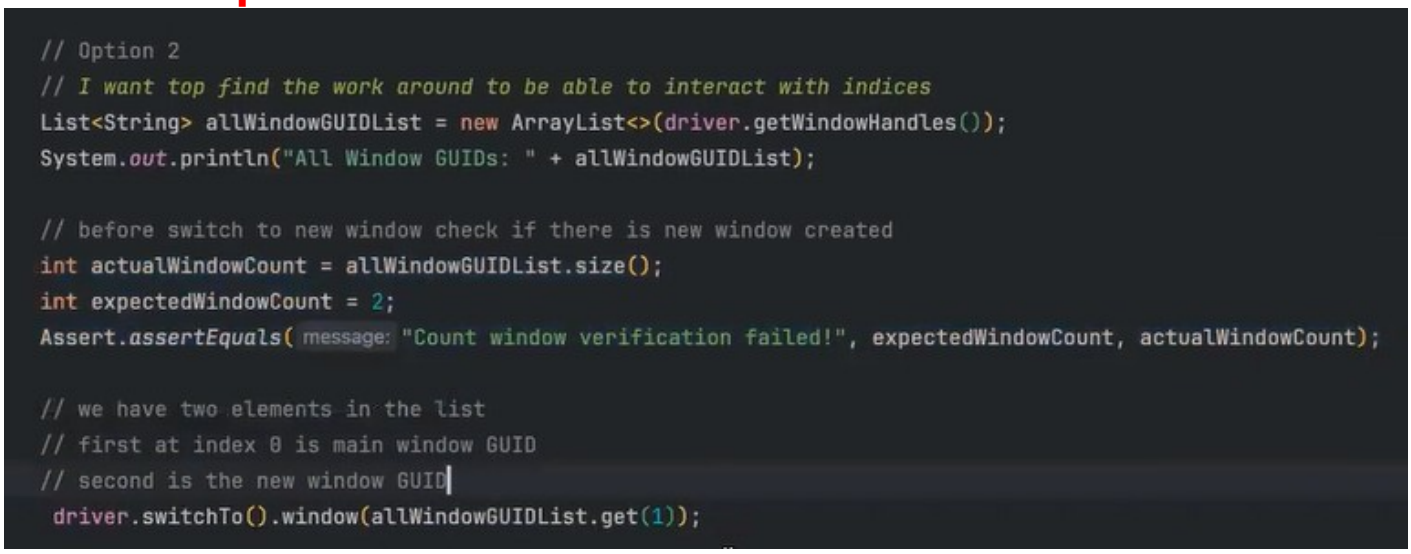
The `qa_configs.properties` file is a lightweight, flexible, and centralized way to manage test settings such as browser type, user credentials, and other configurations. It simplifies test management, increases reusability, and reduces hardcoding in test scripts.

```
package step_definitions;

import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import java.util.Set;
import pages.DemoQaBrowserWindowPage;
import utilities.DriverFactory;

public class DemoQaBrowserWindowPageSteps {

    private WebDriver driver = DriverFactory.getDriver(); // Fixed
instantiation
    private DemoQaBrowserWindowPage demoQaBrowserWindowPage = new
DemoQaBrowserWindowPage(driver); // Corrected instantiation
```

```
    private String mainWindowGUID;

    @Given("I land on Demo QA Browser Window Page")
    public void i_land_on_demo_qa_browser_window_page() {
        String url = "https://demoqa.com/browser-windows";
        driver.get(url);
    }

    @Then("I can retrieve the GUID of the current window")
    public void i_can_retrieve_the_guid_of_the_current_window() {
        // To retrieve the current window GUID we use getWindowHandle()
Selenium method
        mainWindowGUID = driver.getWindowHandle();
        System.out.println("The main window GUID is: " +
mainWindowGUID);
    }

    @When("I click on New Window button")
    public void i_click_on_new_window_button() throws
InterruptedException {
        WebElement newWindowButton =
demoQaBrowserWindowPage.getNewWindowButton(); // Use proper getter for
the button
        newWindowButton.click();
        Thread.sleep(5000);
    }

    @Then("a new window is created and I switch to it")
    public void a_new_window_is_created_and_i_switch_to_it() {
        // Retrieve all window GUIDs
        Set<String> allWindowGUIDs = driver.getWindowHandles();
        System.out.println("All Window GUIDS: " + allWindowGUIDs);

First option:
        // Iterate over the window handles and switch to the new one
        for (String windowGUID : allWindowGUIDs) {
            if (!windowGUID.equals(mainWindowGUID)) {
                driver.switchTo().window(windowGUID);
                System.out.println("Switched to the window GUID: " +
windowGUID);
                break;
            }
        }

    }
```

```
    @Then("I can see the content on the new window as {string}")
    public void i_can_see_the_content_on_the_new_window_as(String
expectedContentText) {
        WebElement contentElementNewWindow =
demoQaBrowserWindowPage.getContentElementNewWindow(); // Use proper
getter
        String actualContentText = contentElementNewWindow.getText();

        Assert.assertEquals("Content verification of new window
failed!", expectedContentText, actualContentText);
    }
}
```

Second Option:


// Option 2
// I want top find the work around to be able to interact with indices
List<String> allWindowGUIDList = new ArrayList<>(driver.getWindowHandles());
System.out.println("All Window GUIDs: " + allWindowGUIDList);
// before switch to new window check if there is new window created
int actualWindowCount = allWindowGUIDList.size();
int expectedWindowCount = 2;
Assert.assertEquals(message: "Count window verification failed!", expectedWindowCount, actualWindowCount);
// we have two elements in the list
// first at index 0 is main window GUID
// second is the new window GUID
driver.switchTo().window(allWindowGUIDList.get(1));

This code defines the **step definitions** for a Cucumber test scenario that interacts with the **Demo QA Browser Window Page**. Each step in the Gherkin feature file (like the one you shared earlier) corresponds to a method in this class. It integrates Selenium WebDriver for browser automation, uses the Page Object Model (POM) for abstraction, and includes test validations.

Here's a detailed breakdown of the code:

---

1. Package Declaration

```
package step_definitions;
```

- Groups this class under the `step_definitions` package, which typically contains all step definitions for Cucumber tests.

---

2. Imports

```
import io.cucumber.java.en.Given;
import io.cucumber.java.en.Then;
import io.cucumber.java.en.When;
import org.junit.Assert;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.WebElement;
import java.util.Set;
import pages.DemoQaBrowserWindowPage;
import utilities.DriverFactory;
```

- **Key Libraries:**
  - `io.cucumber.java.en.Given`, `@Then`, `@When`: Cucumber annotations to map Gherkin steps to Java methods.
  - `org.junit.Assert`: Used for assertions to validate test outcomes.
  - `org.openqa.selenium.WebDriver`, `WebElement`: Selenium WebDriver classes for browser automation.
  - `pages.DemoQaBrowserWindowPage`: The Page Object class used to interact with the "Demo QA Browser Window Page."
  - `utilities.DriverFactory`: A utility class that provides the WebDriver instance.

---

3. Class Fields

```
private WebDriver driver = DriverFactory.getDriver();
private DemoQaBrowserWindowPage demoQaBrowserWindowPage = new
DemoQaBrowserWindowPage(driver);
private String mainWindowGUID;
```

- **`driver`:**
  The WebDriver instance is fetched using the `DriverFactory.getDriver()` method, ensuring that the driver is initialized and shared across the test.
- **`demoQaBrowserWindowPage`:**
  Instantiates the POM class, which contains locators and methods for interacting with the "Demo QA Browser Window Page."
- **`mainWindowGUID`:**
  A string variable to store the GUID (handle) of the main window. It helps differentiate the main window from any newly opened windows.

---

4. Step Definitions

Each method is annotated with a Cucumber step (`@Given`, `@When`, or `@Then`) and defines the behavior of that step.

Step 1: Open the Demo QA Browser Window Page

```
@Given("I land on Demo QA Browser Window Page")
public void i_land_on_demo_qa_browser_window_page() {
    String url = "https://demoqa.com/browser-windows";
    driver.get(url);
}
```

- Opens the specified URL in the browser using the WebDriver's `get` method.

Step 2: Retrieve the Main Window GUID

```
@Then("I can retrieve the GUID of the current window")
public void i_can_retrieve_the_guid_of_the_current_window() {
    mainWindowGUID = driver.getWindowHandle();
    System.out.println("The main window GUID is: " + mainWindowGUID);
}
```

- Fetches the **window handle** of the currently focused browser window using `getWindowHandle()` and stores it in the `mainWindowGUID`.

Step 3: Click the "New Window" Button

```
@When("I click on New Window button")
public void i_click_on_new_window_button() throws InterruptedException {
    WebElement newWindowButton = demoQaBrowserWindowPage.getNewWindowButton();
    newWindowButton.click();
    Thread.sleep(5000); // Pause to allow the new window to open
}
```

- Uses the Page Object class method `getNewWindowButton()` to locate the button and simulate a click.
- Introduces a **`Thread.sleep(5000)`** for simplicity (though `WebDriverWait` is better practice for handling dynamic page loading).

Step 4: Switch to the New Window

```
@Then("a new window is created and I switch to it")
public void a_new_window_is_created_and_i_switch_to_it() {
    Set<String> allWindowGUIDs = driver.getWindowHandles();
    System.out.println("All Window GUIDS: " + allWindowGUIDs);

    for (String windowGUID : allWindowGUIDs) {
        if (!windowGUID.equals(mainWindowGUID)) {
            driver.switchTo().window(windowGUID);
            System.out.println("Switched to the window GUID: " + windowGUID);
            break;
        }
    }
}
```

- **Fetches all window handles:**
  Uses `getWindowHandles()` to retrieve all window GUIDs (main + new windows).
- **Switches to the new window:**
  Loops through the GUIDs, comparing them with `mainWindowGUID` to find the new one, and switches to it using `driver.switchTo().window(windowGUID)`.

Step 5: Validate Content in the New Window

```
@Then("I can see the content on the new window as {string}")
public void i_can_see_the_content_on_the_new_window_as(String expectedContentText) {
    WebElement contentElementNewWindow =
demoQaBrowserWindowPage.getContentElementNewWindow();
    String actualContentText = contentElementNewWindow.getText();

    Assert.assertEquals("Content verification of new window failed!",
expectedContentText, actualContentText);
}
```

- Retrieves the text from the content element of the new window using `getContentElementNewWindow()`.
- Compares it with the expected text passed as a parameter using JUnit's `Assert.assertEquals`.

Key Concepts Illustrated in This Code

1. **Cucumber Integration:**
   The methods map directly to Gherkin steps, enabling natural-language test descriptions.

2. **Page Object Model (POM):**
   The `DemoQaBrowserWindowPage` class abstracts the element locators and actions, making the steps cleaner.
3. **Window Management in Selenium:**
   Demonstrates working with multiple windows using `getWindowHandle()` and `getWindowHandles()`.
4. **Assertions:**
   Uses JUnit assertions to validate the content in the new window.

Improvements:

1. Replace `Thread.sleep()` with **explicit waits** for better reliability:

   ```
   new WebDriverWait(driver, Duration.ofSeconds(10))
       .until(ExpectedConditions.numberOfWindowsToBe(2));
   ```

2. Add exception handling for cases where the new window doesn't open or expected content is missing.

- We know briefly about javascript
  - A **Javascript** is a small chunks of program that makes a website interactive.
  - It can change the content of the html page dynamically.
  - Javascript can listen to even on the web page like reacting to a click on a button, reacting to when checking a checkbox, or when we enter any value into the text bar.
- We need Javascript Executor to run javascript code in selenium.
- Javascript executor allows you to **run pure Javascript code** irrespective of the Selenium language. *(like java,python, ruby etc)*

Javascript Methods

- We should go for Javascriptexecutor only when we are not able to perform a particular task with our selenium *(like not being able to click an element.)*
  - javascriptexecutor is modifying element attributes to get us what we need
- Normally if a button is not clickable, we shouldn't force to find ways to click.
  - But in some cases, we know that button is clickable it's just selenium not able click. This is when we use it.
- There are 2 abstract methods present in Javascript Executor:
  - **executeScript()** → we will be using this method mostly.
  - executeAsyncScript()

JavascriptExecutor Click

There are several actions we can do with help of Javascript Executor

- Click an element
  - `WebElement button =driver.findElement(By.name("btnLogin"));`
  - `JavascriptExecutor js = (JavascriptExecutor)driver;`
  - `js.executeScript("arguments[0].click();", button);`
  - This click is mostly used in places where normal .click() method is not working.
  - **It is effective** → can click on buttons that are not currently visible in screen without scrolling.
    - Regular selenium click is not able to handle such elements. And gives the Exception: ElementNotClickable

JAVASCRIPT SCROLL

Scroll page

- We can scroll the web page, by using JavascriptExecutor:
    - To scroll to the a specific element
    - `js.executeScript("arguments[0].scrollIntoView(true);", element);`
    - To scroll to the end of the page:
    - `js.executeScript("window.scrollTo(0,document.body.scrollHeight)");`